Binary Search Trees

CMPT 225
Objectives

- Understand tree terminology
- Understand and implement tree traversals
- Define the binary search tree property
- Implement binary search trees
- Implement the TreeSort algorithm
Tree Terminology
A set of nodes (or vertices) with a single starting point called the root.

Each node is connected by an edge to another node.

A tree is a connected graph with a path to every node in the tree.

A tree has one less edge than the number of nodes.
Is it a Tree?

- **yes!**
- **NO!** All the nodes are not connected
- **yes!** (but not a binary tree)
- **NO!**
  - There is an extra edge (5 nodes and 5 edges)
- **yes!** (it’s actually a similar graph to the blue one)
Node $v$ is said to be a *child* of $u$, and $u$ the *parent* of $v$ if
- There is an edge between the nodes $u$ and $v$, and
- $u$ is above $v$ in the tree,

This relationship can be generalized
- E and F are *descendants* of A
- D and A are *ancestors* of G
- B, C and D are *siblings*
- F and G are?
More Tree Terminology

- A leaf is a node with no children
- A path is a sequence of nodes $v_1 \ldots v_n$
  - where $v_i$ is a parent of $v_{i+1}$ ($1 \leq i \leq n$)
- A subtree is any node in the tree along with all of its descendants
- A binary tree is a tree with at most two children per node
  - The children are referred to as left and right
  - We can also refer to left and right subtrees
Tree Terminology Example

C, E, F and G are leaves

subtree rooted at B

path from A to D to G
Measuring Trees

- The *height* of a node $v$ is the length of the longest path from $v$ to a leaf
  - The height of the tree is the height of the root
- The *depth* of a node $v$ is the length of the path from $v$ to the root
  - This is also referred to as the *level* of a node
- Note that there is a slightly different formulation of the height of a tree
  - Where the height of a tree is said to be the number of different *levels* of nodes in the tree (including the root)
Height of a Binary Tree

height of node B is 2

height of the tree is 3

depth of node E is 2

level 1

level 2

level 3
A binary tree is **perfect**, if
- No node has only one child
- And all the leaves have the same depth

A perfect binary tree of height $h$ has
- $2^{h+1} - 1$ nodes, of which $2^h$ are leaves

Perfect trees are also **complete**
Each level doubles the number of nodes
- Level 1 has 2 nodes \((2^1)\)
- Level 2 has 4 nodes \((2^2)\) or 2 times the number in Level 1

Therefore a tree with \(h\) levels has \(2^{h+1} - 1\) nodes
- The root level has 1 node

The bottom level has \(2^h\) nodes, that is, just over \(\frac{1}{2}\) the nodes are leaves.
A binary tree is *complete* if

- The leaves are on at most two different levels,
- The second to bottom level is completely filled in and
- The leaves on the bottom level are as far to the left as possible
A binary tree is *balanced* if
- Leaves are all about the same distance from the root
- The exact specification varies

Sometimes trees are balanced by comparing the height of nodes
- e.g. the height of a node’s right subtree is at most one different from the height of its left subtree

Sometimes a tree's height is compared to the number of nodes
- e.g. red-black trees
Balanced Binary Trees
Unbalanced Binary Trees

A

B
C
D
E
F

A

B

C
D
Tree Traversals
A traversal algorithm for a binary tree visits each node in the tree
- Typically, it will do something while visiting each node!
- Traversal algorithms are naturally recursive
- There are three traversal methods
  - Inorder
  - Preorder
  - Postorder
The visit function would do whatever the purpose of the traversal is, for example print the data in the node.
visit(nd)
preOrder(nd->leftChild)
preOrder(nd->rightChild)
postOrder(nd->leftChild)
postOrder(nd->rightChild)
visit(nd)
Binary Search Trees
The binary tree ADT can be implemented using different data structures

- Reference structures (similar to linked lists)
- Arrays

Example implementations

- Binary search trees (references)
- Red – black trees (references again)
- Heaps (arrays) – not a binary search tree
- B trees (arrays again) – not a binary search tree
Consider maintaining data in some order

- The data is to be frequently searched on the sort key e.g. a dictionary

Possible solutions might be:

- A sorted array
  - Access in $O(\log n)$ using binary search
  - Insertion and deletion in linear time
- An ordered linked list
  - Access, insertion and deletion in linear time
Dictionary Operations

- The data structure should be able to perform all these operations efficiently
  - Create an empty dictionary
  - Insert
  - Delete
  - Look up
- The insert, delete and look up operations should be performed in at most $O(\log n)$ time
A binary search tree is a binary tree with a special property:

- For all nodes in the tree:
  - All nodes in a left subtree have labels less than the label of the subtree's root
  - All nodes in a right subtree have labels greater than or equal to the label of the subtree's root

Binary search trees are fully ordered.
BST Example
An inorder traversal retrieves the data in sorted order

inOrder(nd->leftChild)
visit(nd)
inOrder(nd->rightChild)
Binary Search Tree Search
Binary search trees can be implemented using a reference structure
Tree nodes contain data and two pointers to nodes

Node* leftChild  data  Node* rightChild

- data to be stored in the tree (usually an object)
- references or pointers to Nodes
To find a value in a BST search from the root node:
- If the target is less than the value in the node search its left subtree
- If the target is greater than the value in the node search its right subtree
- Otherwise return true, (or a pointer to the data, or ...)

How many comparisons?
- One for each node on the path
- Worst case: height of the tree + 1
BST Search Example

Click on a node to show its value.
bool search(Node* nd, int x)
{
    if (nd == NULL)
    {
        return false;
    } else if (x == nd->data)
    {
        return true;
    } else if (x < nd->data)
    {
        return search(x, nd->left);
    } else {
        return search(x, nd->right);
    }
}

reached the end of this path

note the similarity to binary search

called by a helper method like this:
search(root, target)
BST Insertion
The BST property must hold after insertion
Therefore the new node must be inserted in the correct position
  - This position is found by performing a search
  - If the search ends at the NULL left child of a node make its left child refer to the new node
  - If the search ends at the NULL right child of a node make its right child refer to the new node
- The cost is about the same as the cost for the search algorithm, $O(\text{height})$
**BST Insertion Example**

1. **Insert 43**
2. **Create new node**
3. **Find position**
4. **Insert new node**

Diagram:

```
    47
   /   
32    63
 /     / 
19    54  79
|      |    |
10    41  59
|  37   |  |
7  12   43  96
|    30   |
7  12
```

John Edgar
BST Deletion
BST Deletion

- After deletion the BST property must hold
- Deletion is not as straightforward as search or insertion
  - With insertion the strategy is to insert a new leaf
  - Which avoids changing the internal structure of the tree
  - This is not possible with deletion
    - Since the deleted node's position is not chosen by the algorithm
- There are a number of different cases to be considered
BST Deletion Cases

- The node to be deleted has no children
  - Remove it (assigning NULL to its parent’s reference)
- The node to be deleted has one child
  - Replace the node with its subtree
- The node to be deleted has two children
  - ...
BST Deletion – Target is a Leaf

delete 30
delete 79
replace with subtree
BST Deletion – Target Has One Child

delete 79 after deletion
One of the issues with implementing a BST is the necessity to look at both children.
- And, just like a linked list, *look ahead* for insertion and deletion.
- And check that a node is null before accessing its member variables.

Consider deleting a node with one child in more detail.
delete 59

Step 1 - we need to find the node to delete and its parent

it’s useful to know if nd is a left or right child

while (nd != target)
    if (nd == NULL)
        return
    if (target < nd->data)
        parent = nd
        nd = nd->left
        isLeftChild = true
    else
        parent = nd
        nd = nd->right
        isLeftChild = false
delete 59

Now we have enough information to detach 59 and attach its child to 54.
The most difficult case is when the node to be deleted has two children

- The strategy when the deleted node had one child was to replace it with its child
- But when the node has two children problems arise

Which child should we replace the node with?

- We could solve this by just picking one ...

But what if the node we replace it with also has two children?

- This will cause a problem
Deleted Node Has 2 Children

- delete 32
- let's say that we decide to replace it with its right child (41)

But 41 has 2 children, and it also has to inherit (adopt?) the other child of its deleted parent.
When a node has two children, instead of replacing it with one of its children find its predecessor

- A node’s predecessor is the right most node of its left subtree
- The predecessor is the node in the tree with the largest value less than the node’s value

- The predecessor cannot have a right child and can therefore have at most one child
  - Why?
32’s predecessor

The predecessor of 32 is the right most node in its left subtree

The predecessor cannot have a right child as it wouldn’t then be the right most node
The predecessor has some useful properties

- Because of the BST property it must be the largest value less than its ancestor’s value
  - It is to the right of all of the nodes in its ancestor’s left subtree so must be greater than them
  - It is less than the nodes in its ancestor’s right subtree
- It can have only one child
- These properties make it a good candidate to replace its ancestor
What About the Successor?

- The successor to a node is the *left* most child of its *right* subtree
  - It has the *smallest* value *greater* than its ancestor’s value
  - And cannot have a left child
- The successor can also be used to replace a deleted node
  - Pick either the predecessor or successor!
Deleted Node Has 2 Children

delete 32
find successor and detach

diagram with nodes 47, 32, 63, 19, 37, 41, 54, 10, 23, 30, 44, 53, 59, 96, 7, 12, 91, 97

temp
**Deleted Node Has 2 Children**

1. **delete 32**
2. **find successor and detach**
3. **attach node’s children to its successor**
Deleted Node Has 2 Children

delete 32
find successor and detach
attach node’s children
make successor child of node’s parent
Deleted Node Has 2 Children

- delete 32
- find successor and detach
- attach node’s children
- make successor child

In this example, the successor had no subtree.
Deleted Node Has 2 Children

delete 63
find predecessor*

*just because ...

temp
Deleted Node Has 2 Children

delete 63
find predecessor
attach predecessor’s subtree to its parent

```
        47
       /  \
     32   63
    / \   /  \
  19  41 54  \\
 /   /  \
10  23  44  53
 |   |    |   |
7   37  53  96
  |       |
  12  30  57
     /
      |
     7
```
temp
Deleted Node Has 2 Children

- delete 63
- find predecessor
- attach pre’s subtree
- attach node’s children to predecessor

```
19
  10 23 37 44
    7 12 30
        96
```
```
32
  41
    30
```
```
47
  63
    54
      59
      96
```
```
59
  57
    91 97
```

(temp)
Deleted Node Has 2 Children

- delete 63
- find predecessor
- attach pre’s subtree
- attach node’s children
- attach the predecessor to the node’s parent

![Diagram of a binary tree with nodes and arrows indicating the operations to be performed when deleting node 63.](diagram.png)
Deleted Node Has 2 Children

- delete 63
- find predecessor
- attach pre’s subtree
- attach node’s children
- attach the predecessor to the node’s parent
Instead of deleting a BST node mark it as deleted in some way

- Set the data object to `null`, for example

And change the insertion algorithm to look for empty nodes

- And insert the new item in an empty node that is found on the way down the tree
An alternative to the deletion approach for nodes with 2 children is to replace the data:

- The data from the predecessor node is copied into the node to be deleted
- And the predecessor node is then deleted
  - Using the approach described for deleting nodes with one or no children

This avoids some of the complicated pointer assignments.
BST Efficiency
The efficiency of BST operations depends on the *height* of the tree

- All three operations (search, insert and delete) are $O(\text{height})$
- If the tree is complete the height is $\lceil \log(n) \rceil$
  - What if it isn’t complete?
Height of a BST

- Insert 7
- Insert 4
- Insert 1
- Insert 9
- Insert 5
- It’s a complete tree!

height = \lfloor \log(5) \rfloor = 2
Height of a BST

- Insert 9
- Insert 1
- Insert 7
- Insert 4
- Insert 5
- It’s a linked list with a lot of extra pointers!

height = n-1 = 4 = O(n)
Balanced BSTs

- It would be ideal if a BST was always close to complete
  - i.e. balanced
- How do we guarantee a balanced BST?
  - We have to make the structure and / or the insertion and deletion algorithms more complex
    - e.g. red – black trees.
It is possible to sort an array using a binary search tree

- Insert the array items into an empty tree
- Write the data from the tree back into the array using an InOrder traversal

Running time = \( n \times (\text{insertion cost}) + \text{traversal} \)

- Insertion cost is \( O(h) \)
- Traversal is \( O(n) \)
- Total = \( O(n) \times O(h) + O(n) \), i.e. \( O(n \times h) \)
- If the tree is balanced = \( O(n \times \log(n)) \)